PROBLEM SET 1

DATA STRUCTURES AND ALGORITHMS 2 - FALL 2023

yl2nr

DUE SEPTEMBER 8, 2023 AT 11:59 PM

Collaboration Policy: You are encouraged to collaborate with up to 4 other students, but all work submitted must be your own *independently* written solution. List the computing ids of all of your collaborators in the `collabs` command at the top of the tex file. Do not share written notes, documents (including Google docs, Overleaf docs, discussion notes, PDFs), or code. Do not seek published or online solutions for any assignments. If you use any published or online resources (which may not include solutions) when completing this assignment, be sure to cite them. Do not submit a solution that you are unable to explain orally to a member of the course staff. Any solutions that share similar text/code will be considered in breach of this policy. Please refer to the syllabus for a complete description of the collaboration policy.

Collaborators: list your collaborators

Sources: list your sources

PROBLEM 1 *Order class proof using limits*

Prove that $n^k \in o(a^n)$ for $a > 1$. Use the limit definition of the order class. If you find you need to use L'Hôpital's rule, note that $(a^n)' = (\log a)a^n$.

Solution:

Proof.

We want to show $\exists m > 0 \ s.t \ \forall n > k, \ n^k < a^n \iff \lim_{n\to\infty} \frac{n^k}{a^n} = 0$ as by the definition of a sequence being convergence, if $\lim_{n\to\infty} \frac{n^k}{a^n} = 0$, then we are sure that $\exists m > 0 \ s.t \ \forall n > m$, $\frac{n^k}{a^n} < 1 \implies n^k < a^n$.

$$\lim_{n\to\infty} \frac{n^k}{a^n} = \lim_{n\to\infty} \frac{\frac{dn^k}{dn}}{\frac{da^n}{dn}} = \lim_{n\to\infty} \frac{kn^k}{(\log a)a^n}$$

As both the nominator and denominator approach infinity as n approaches infinity, we applied L'Hôpital's rule. We repeat to apply L'Hôpital's rule to the sequence for k times until the nominator becomes a constant.

$$\lim_{n\to\infty} \frac{n^k}{a^n} = \lim_{n\to\infty} \frac{\frac{dn^k}{dn}}{\frac{da^n}{dn}} = \lim_{n\to\infty} \frac{kn^k}{(\log a)a^n} = \lim_{n\to\infty} \frac{\frac{dkn^k}{dn}}{\frac{d\log a)a^n}{dn}} = ... = \lim_{n\to\infty} \frac{k!}{(\log a)^k a^n} = \frac{k!}{(\log a)^k} \cdot \lim_{n\to\infty} \frac{1}{a^n} = 0$$

Thus, we have shown that $\lim_{n\to\infty} \frac{n^k}{a^n} = 0$, and by definition of the convergence, $\exists m > 0 \ s.t \ \forall n > m$, $\frac{n^k}{a^n} < 1 \implies n^k < a^n$. As we proved after a certain value of n, a^n grows strictly slower than $a^n, forsomea > 1$, $n^k \in o(a^n)$ for $a > 1$. □

PROBLEM 2 *FlyMe Airlines*

An airline, FlyMe Airlines, is analyzing their network of airport connections. They have a graph $G = (A, E)$ that represents the set of airports A and their flight connections E between them. They define $hops(a_i, a_k)$ to be the smallest number of flight connections between two airports. They define $maxHops(a_i)$ to be the number of hops to the airport that is farthest from a_i, i.e. $maxHops(a_i) = max(hops(a_i, a_j)) \, \forall a_j \in A$.

The airline wishes to define one or more of their airports to be "Core 1 airports." Each Core 1 airport a_i will have a value of $maxHops(a_i)$ that is no larger than any other airport. You can think of the Core 1 airports as being "in the middle" of FlyMe Airlines' airport network. The worst flight from a Core 1 airport (where "worst" means having a large number of connections) is the same or better than any other airport's worst flight connection (i.e. its $maxHops()$ value).

They also define "Core 2 airports" to be the set of airports that have a $maxHops()$ value that is just 1 more than that of the Core 1 airports. (Why do they care about all this? Delays at Core 1 or Core 2 airports may have big effects on the overall network performance.)

Your problem: Describe an algorithm that finds the set of Core 1 airports and the set of Core 2 airports. Give its time-complexity. The input is $G = (A, E)$, an undirected and unweighted graph, where $e = (a_i, a_j) \in E$ means that there is a flight between a_i and a_j. Base your algorithm design on algorithms we have studied in this unit of the course.

Solution:

The general idea is to run BFS from each vertex (airport) to find the $maxHops()$ of each vertex (airport). As we traverse all airport, we dynamically keep a minimum and second least value of $maxHops(a_i)$ and return the corresponding Core 1 airports and Core 2 airports when we finish running BFS from all vertexes (airports).

```
def find_core_airports(A,E):
    minimum = INT_MAX
    second_least = INT_MAX
    core_1 = []
    core_2 = []
    for a in A:
      maxHop = maxHop_bfs(a,A,E)
      if maxHop < minimum:
          second_least = minimum
          minimum = maxHop
          core_2 = core_1
          core_1 = [a]
      elif maxHop == minimum:
          core_1.append(a)
      elif maxHop < second_least:
          second_least = maxHop
          core_2 = [a]
      elif maxHop == second_least:
          core_2.append(a)
    return {"core_1 airports" : core_1, "core_2 airports" : core_2}

def maxHop_bfs(start,A,E):
    maxHop = 0
    to_visit = [start]
    hops = [-1] * number of vertexes in A
    visited = [False] * number of vertexes in A
```

```
visited[start] = True
hops[start] = 0
while to_visit:
      current = to_visit.pop[0]
      for node in current.neighbor():
         if visited[node] == False:
            to_visit.append(node)
            visited[node] = True
            hops[node] = hops[current] + 1
            maxHop = max(hops[node], maxHop)
return maxHop
```

The time complexity of each BFS is $O(A + E)$.
Thus, the total time complexity is $O(A(A + E))$.

PROBLEM 3 *Counting Shortest Paths*

Given a graph $G = (V, E)$, and a starting node s, let $\ell(s, t)$ be the length of the shortest path in terms of number of edges between s and t. Give a clear description of an algorithm that computes the number of distinct paths from s to t that have length exactly $\ell(s, t)$.

Solution:

The algorithm will be a BFS with one additional list to keep track of the number of shortest paths from the start vertex to every other vertex connected.

```
def num_shortest_path(G, s, t):

    layer = 0
    numOfPath = 0
    depth = [INT_MAX] * number of vertexes in G
    numPath = [0] * number of vertexes in G
    visited = [False] * number of vertexes in G
    to_visit = [s]
    depth[s] = 0
    numPath[s] = 1
    visited[current] = True

    while to_visit:
      current = to_visit.pop[o]
      layer = depth[current]
      numOfPath = numPath[current]
      for node in current.neighbor():

        if visited[node] == False:
          to_visit.append(node)
          visited[node] = True

        if layer + 1 < depth[node]:
          depth[node] = layer + 1
          numPath[node] = numOfPath

        if layer + 1 == depth[node]:
          numPath[node] = numPath[node] + numOfPath

    return numPath[t]
```

PROBLEM 4 *Coffee*

Charlottesville is known for some of its locally-roasted coffees, each with its own unique flavor combination. Suppose a group of coffee enthusiasts are given n samples $c_1, c_2, ..., c_n$ of freshly-brewed Charlottesville coffee by a coffee-skeptic. Each sample is either a *Milli* roast or a *Shenandoah Joe* roast. The enthusiasts are given each pair (c_i, c_j) of coffees to taste, and they must collectively decide whether: (a) both are the same brand of coffee, (b) they are from different brands, or (c) they cannot agree (i.e., they are unsure whether the coffees are the same or different). Note: all n^2 pairings are tested, including $(c_i, c_i), (c_i, c_j)$, and (c_j, c_i), but not all pairs have "same" or "different" decisions.

At the end of the tasting, suppose the coffee enthusiasts have made m judgments of "same" or "different." Give an algorithm that takes these m judgments and determines whether they are *consistent*. The m judgments are *consistent* if there is a way to label each coffee sample c_i as either *Shenandoah Joe* or *Milli* such that for every taste-comparison (c_i, c_j) labeled "same," both c_i and c_j have the same label, and for every taste-comparison labeled "different," both c_i and c_j are labeled differently. Your algorithm should run in $O(m + n)$ time. Prove the correctness and running time of your algorithm. Note: you do *not* need to determine the brand of each sample c_i, only whether the coffee enthusiasts are consistent in their labelings.

Solution:

1. Create a dictionary or hashmap to store vertexes and edges for a graph.

2. Start to go over m judgments. For each judgment (c_i, c_j), if it is labeled as same, then:
If both c_i, c_j are not in the keys of the dictionary, add them to the dictionary as vertexes in the graph.
If one of them is in the dictionary, then just add the other one in as a key of the dictionary, a vertex in the graph, but not adding any edge.
If both are already in the keys of the dictionary (vertexes are created), then do nothing.

3. If (c_i, c_j) is labeled as different, then:
If both c_i, c_j are not in the keys of the dictionary, add them as keys. Also, add edges (c_i, c_j) and (c_j, c_i) to the graph.
If one of them is in the dictionary, then just add the other one in as a key of the dictionary, a vertex in the graph, Also, add edges (c_i, c_j) and (c_j, c_i) to the graph.
If both are already in the keys of the dictionary (vertexes are created), then just add edges (c_i, c_j) and (c_j, c_i) to the graph.

4. After creating the graph according m judgments, we use BFS to determine if the graph is Bipartite. If it is Bipartite, then return *consistent*, otherwise, return *not consistent*.

As the time complexity of add vertexes or edges is $O(1)$, so go over m pairs and add accordingly has time complexity $O(m)$. The BFS has time complexity of $O(n + m)$ as there can be as many as n vertexes and m pairs can generate as many as m edges.
Thus the final time complexity is $O(n + 2m) = O(n + m)$.

PROBLEM 5 *Ancient Population Study*

Historians are studying the population of the ancient civilization of *Algorithmica*. Unfortunately, they have only uncovered incomplete information about the people who lived there during Algorithmica's most important century. While they do not have the exact year of birth or year of death for these people, they have a large number of possible facts from ancient records that say when a person lived relative to when another person lived.

These possible facts fall into two forms:

- The first states that one person died before the another person was born.
- The second states that their life spans overlapped, at least partially.

The Algorithmica historians need your help to answer the following questions. First, is the large collection of uncovered possible facts internally consistent? This means that a set of people could have lived with birth and death years that are consistent with all the possible facts they've uncovered. (The ancient records *may not be accurate*, meaning all the facts taken together cannot possibly be true.) Second, if the facts are consistent, find a sequence of birth and death years for all the people in the set such that all the facts simultaneously hold. (Examples are given below.)

We'll denote the n people as $P_1, P_2, \ldots, P_n$. For each person P_i, their birth-year will be b_i and their death-year will be d_i. (Again, for this problem we do not know and cannot find the exact numeric year value for these.)

The possible facts (input) for this problem will be a list of relationships between two people, in one of two forms:

- P_i *prec* P_j (indicates P_i died before P_j was born)
- P_i *overlaps* P_j (indicates their life spans overlapped)

If this list of possible facts is not consistent, your algorithm will return "not consistent". Otherwise, it will return a possible sequence of birth and death years that is consistent with these facts.

Here are some examples:

- The following facts about $n = 3$ people are **not** consistent: P_1 *prec* P_2, P_2 *prec* P_3, and P_3 *prec* P_1.
- The following facts about $n = 3$ people **are** consistent: P_1 *prec* P_2 and P_2 *overlaps* P_3. Here are two possible sequences of birth and death years:
  $b_1, d_1, b_2, b_3, d_2, d_3$
  $b_1, d_1, b_3, b_2, d_2, d_3$
  (Your solution only needs to find one of any of the possible sequences.)

Your answer should include the following. Clearly and precisely explain the graph you'll create to solve this problem, including what the nodes and edges will be in the graph. Explain how you'll use one or more of the algorithms we've studied to solve this graph problem, and explain why this leads to a correct answer. Finally, give the time-complexity of your solution.

Solution:

The general idea here is to build a directed graph so that each person's death d_i is a vertex, and each person's birth b_i is another vertex. All vertexes are connected accordingly. Then we use DFS to do topological sort. If there is any cycle or backward edge, return *inconsistent*.

As we go through the inputs,

- If P_i *prec* P_j (indicates P_i died before P_j was born), we created four corresponding vertexes if they are not already in the graph. Then we add edges (b_i,d_i), (d_i,b_j), and (b_j,d_j). So we connect the birth to death of each person himself, and connect the death of the precedent to the birth of the decedent.

- If P_i *overlaps* P_j (indicates their life spans overlapped), then we add edges (b_i, b_j), (b_i,d_i), (b_j, d_j), (d_j,d_i). So we connect first person's birth to the second person's birth, and second person's death to the first person's death.

Then we apply DFS_sweep to the graph, and do *topological sort* to produce the sequence of birth and death.

```
def top_sort(V,E): #V is collection of vertexes; E is collection of edges.
    visited = [False] * number of vertexes
    finished = []
      for p in V:
        if visited[p] == False:
          finish_time(V,E,p,visited,finished)
    return reverse(finished)

def finish_time(V,E,current,visited,finished):
    visited[current] = True
    for v in neighbors(current):
      if visited[v] == True and v not in finished:
        # Check for cycle
        return "Inconsistent"
      elif visited[v] == False:
        finish_time(V,E,v,visited,finished)
    finished.append(current)
```

The time complexity for creating the graph is $O(2n)$ as we need to add as many as $2 * n$ vertexes (2 for each person).

The time complexity for DFS_sweep would be $O(2n + E)$ and E is the number of edges.

Thus, the total time complexity is $O(4n + E)$.